Programmazione 1

Enrico Bragastini

3 giugno 2021

Indice

Stringhe

Una stringa è un *array di caratteri* terminato dal carattere '\0'. Quindi dichiarando una stringa lunga 20 caratteri, saranno 19+1 dove l'ultimo è il carattere di terminazione.

1.1 Dichiarazione di una stringa

Dichiarazione di una stringa come array di lunghezza 20 caratteri:

```
char stringa[20];
```

Dichiarazione di una stringa e inizializzazione del suo contenuto:

```
char stringa[20] = "ciao, tutto bene?"; # 17+1 caratteri
```

in questo caso viene allocato uno spazio sufficiente per 20 caratteri, ma ne vengono utilizzati solamente 18.

1.2 Lettura di una stringa

Con lo **scanf** si può leggere un array di caratteri senza avere bisogno di un ciclo. Viene inserito automaticamente un '\0' alla fine della stringa. Leggerà fino a trovare uno spazio oppure un andata a capo.

```
scanf("%s", stringa);
```

Per leggere stringhe anche con spazi o con andate a capo si può utilizzare gets. Gets è pericoloso perché può leggere anche stringhe più lunghe dello spazio allocato.

```
gets(stringa);
```

Il modo corretto è leggere stringhe con spazi è mediante un ciclo:

```
int i;
i = 0;
do{
    scanf("%c", &stringa[i]);
    i++;
} while(stringa[i-1] != '\n');
stringa[i-1] = '\0'
```

1.3 Libreria string.h

Alcune funzioni dalla libreria *string.h* utili per eseguire operazioni con le stringhe:

- STRLEN: restituisce la **lunghezza** di una stringa senza il '\0'
- STRCMP: **compara** due stringhe, restituendo un intero che vale 0 se le stringhe sono uguali, -1 se $S1 < S2$ e 1 se $S1 > S2$.
- STRCPY: **copia** il contenuto di $S2$ in $S1$
- STRCAT: restituisce la **concatenazione** di $S2$ a $S1$

Struct

Le *Struct* sono delle variabili strutturate, ovvero degli aggregati di campi, utilizzate per memorizzare informazioni non omogenee di tipo. Hanno la seguente forma:

```
struct{
    # campi
} <nome>;
```

2.1 Esempio e dimensione di una Struct

Esempio di Struct per memorizzare delle informazioni di uno studente:

```
struct{
    char matricola[6];
    char nome[20+1];
    char cognome[20+1];
    float media;
} studente;
```

La dimensione di questa variabile equivale quindi a

$$6 \times 8 \text{ bit} + 21 \times 8 \text{ bit} + 21 \times 8 \text{ bit} + 1 \times 32 \text{ bit} = 416 \text{ bit}$$

2.2 Usare le struct

Per accedere alle variabili interne di una Struct si usa la seguente sintassi:

```
struct.variabile
```

Esempio di lettura e scrittura:

```
printf("Inserire il voto medio");
scanf("%f", &studente.media);
printf("Voto medio: ", studente.media);

int i;
for(i=0; i<6; i++){
    scanf("%c", &studente.matricola[i]);
}
```

È possibile dichiarare più struct in un colpo solo:

```
struct{
    # campi
} <nome1>, <nome2>;
```

ed è possibile copiare una struct con un'altra come si farebbe per una normale variabile:

```
struct1 = struct2;
```

questo copierà rispettivamente tutti i campi di *struct2* dentro a *struct1*

2.3 Struct per creare nuovi tipi di variabili

Dichiarando una struct come segue:

```
struct info{
    int min;
    int max
};
```

verrà creato un nuovo tipo di variabile. Sarà quindi possibile dichiarare variabili di tipo **info**. Per essere utilizzato il tipo **info**, è necessaria anche la keyword **struct**. Quindi per dichiarare due variabili *a* e *b* di tipo *info*, bisognerà scrivere

```
struct info a, b;
```

2.3.1 Typedef

Esiste in C un'istruzione che permette di creare dei nuovi tipi di variabili, l'istruzione **tupedef**.

```
typedef int mio_int;
```

Scrivendo un'istruzione del genere, creiamo un nuovo tipo, chiamato *mio_int*, che è un *alias* del tipo *int*.

Questa istruzione risulta particolarmente utile per creare delle nuove tipologie di variabili, in combinata con le *struct*. Vediamo un esempio:

```
typedef struct{
    int min;
    int max;
} t_info;
```

Così facendo avremo creato una variabile **user-defined** di tipo *t_info*. Potremo quindi scrivere

```
t_info a, b;
```

per dichiarare due variabili di tipo *t_info*

Puntatori

Un puntatore è una *variabile* che contiene l'**indirizzo** di un'altra variabile.

3.1 Dichiarazione di un puntatore

Un puntatore si dichiara specificando il tipo della variabile a cui si sta puntando e il nome del puntatore stesso:

```
int *p;
```

Supponiamo quindi di avere una variabile intera **a** e il puntatore **p**. Vogliamo che il puntatore contenga l'indirizzo della variabile a:

```
int main(){
    int a;
    int *p;
    a = 5;
    p = &a;      // assegnazione di p
    return 0;
}
```

È possibile ora andare a modificare il valore della variabile **a** accedendovi sia tramite la stessa variabile **a**, che tramite il puntatore **p**:

```
a++;
(*p)++;
```

3.2 Operatori

Per lavorare con i puntatori si utilizzano due operatori:

- *: L'*asterisco* è detto operatore di *deferenziazione*. Permette di lavorare con la variabile puntata dal puntatore.
- &: Il simbolo *E-commerciale* permette di riferirsi all'indirizzo di una variabile.

Esempio:

```
int main(){
    int i = 5, j = 9;
    int *p;
    p = &j;      // p punta alla variabile j
    *p = i;      // viene modificato il valore di j
}
```

3.3 Puntatore a puntatore

È possibile anche creare dei puntatori a dei puntatori.

Esempio:

```
int main(){
    int c, d;
    int *p1;
    int *p2;
    int **p3;
    int **p4;
    ...
}
```

Le variabili create sono:

- **c** e **d**: variabili intere
- **p1** e **p2**: puntatori a variabili intere
- **p3** e **p4**: puntatori a puntatori a variabili intere

Se si andasse ad aggiungere il seguente codice:

```
c = 54;
d = 10;
p1 = &c;
p2 = p1;
printf("%d %d %d %d",c,d,*p1,*p2);
```

verrebbe stampata la stringa "54 10 54 54".

```
p1 = &d;
*p1 = *p1 + *p2:
p3 = &p1;
p4 = &p2;
*p4 = *p3;
printf("%d %d %d %d",c,d,*p1,*p2);
```

ora verrebbe stampata la stringa "54 64 64 64".

3.4 Relazione tra array e indirizzi di memoria

Vediamo alcune relazioni che ci sono tra gli array e gli indirizzi di memoria. Sia dato un array n di 10 interi:

- $n \equiv \&n[0]$
- $n + i \equiv \&n[i]$
- $*n \equiv *(\&n[0]) \equiv n[0]$
- $*(n + i) \equiv n[i]$

File

I file sono strutture di memoria permanenti. Vengono gestiti dal file system del Sistema Operativo. Il S.O. mette a disposizione dei processi la **tabella dei file aperti**, ovvero un *array di strutture*. Ogni cella di questo *array* è di tipo **FILE** e contiene varie informazioni quali nome del file, modalità di accesso (lettura, scrittura, append...) puntatore alla posizione corrente indicatore di fine file...

4.1 Sintassi

Per lavorare con i file è necessario dichiarare un puntatore all'array FILE:

```
FILE * fp;
```

Per **aprire** un file si utilizza l'istruzione fopen:

```
fp = fopen(nomefile, modalita);
```

dove la modalità può variare tra:

- **r**: lettura
- **w**: scrittura
- **a**: append (scrittura dalla fine del file)

È necessario assicurarsi che l'associazione del puntatore sia stata eseguita correttamente:

```
if(fp){
    ...
}
```

Per **scrivere** su file si usa la funzione *fprintf*:

```
fprintf(fp, "%d", num);
```

Per **leggere** da file si usa la funzione *fscanf*:

```
fscanf(fp, "%d", &num);
```

Per **liberare** l'accesso al file è necessario usare la funzione **fclose**:

```
fclose(fp);
```

4.2 End of File

La funzione **int feof(FILE * fp)** è una funzione molto utile per sapere se si è arrivati a leggere la fine del file. La fine del file viene intercettata solamente dopo una lettura, quindi questa chiamata di funzione bisogna farla dopo aver letto almeno una volta dal file. Questa funzione restituisce

- Vero (1): è stata raggiunta la fine del file
- Falso (0): non è stata raggiunta la fine del file

Esempio di lettura da file:

```
int main(){
    int num;
    FILE * fp;
    if(fp){
        fscanf(fp, "%d", &num);
        while(!feof(fp)){
            printf("%d", num);
            fscanf(fp, "%d", &num);
        }
        fclose(fp);
    }
    else printf("Errore in lettura.");
}
```

Allocazione dinamica della memoria

Fino ad ora abbiamo utilizzato una *allocazione statica (automatica)* della memoria. Con tale modalità le variabili vengono allocate automaticamente in memoria nello **stack** quando si entra in un blocco di codice (ad esempio una funzione) e corrispondentemente vengono distrutte automaticamente quando si esce dal blocco stesso.

L'**allocazione dinamica** della memoria consente di determinare lo spazion necessario a certe variabili **durante l'esecuzione** del programma. Una variabile viene allocata dinamicamente in memoria nello **heap** attraverso specifiche istruzioni, e rimane tale finché non viene *esplicitamente* deallocata.

5.1 Struttura della memoria

La memorià è suddivisa nelle seguenti sezioni:

- **Heap**: Zona separata che cresce nel verso opposto allo stack dove vengono allocati dinamicamente degli spazi di memoria di cui si conosce la dimensione solamente durante l'esecuzione del programma. È importante deallocare manualmente lo spazio precedentemente allocato.
- **Stack**: Sezione in cui vengono allocate le variabili locali automatiche. Per prime troviamo le variabili del *main* e successivamente le variabili dei sottoprogrammi chiamati. La dimensione dello stack dipende dal momento di esecuzione che si sta analizzando, questo perché alla fine di un sottoprogramma tutte le variabili non *static* vengono deallocate automaticamente.
- **Global/Static variables**: Sezione in cui vengono caricate le variabili globali, visibili da tutte le funzioni e esterne al *main*, e le variabili statiche (variabili locali di una funzione ma che mantengono il valore durante l'esecuzione)
- **Code**: Spazio di memoria in cui, nella fase di loading della catena di compilazoine del C, vengono caricate le istruzioni da eseguire.

5.2 Funzioni calloc() e malloc()

Le funzioni **calloc()** e **malloc()** della libreria standard <**stdlib.h**> consentono di allocare dinamicamente la memoria.

5.2.1 calloc()

La funzione calloc() richiede due argomenti che specificano: Il numero di elementi da riservare e la dimensione di ciascun elemento in byte. Questa funzione restituisce poi

un puntatore all'inizio dell'area di memoria allocata. Questo puntatore è di tipo *void*, puntatore generico. Per essere allocato alla variabile visogna eseguire un *cast* di tipo.

L'area di memoria allocata viene automaticamente posta a 0.

```
int *int_ptr;
...
int_ptr = (int *) calloc(1000, sizeof(int));
```

5.2.2 malloc()

La funzione malloc() richiede un solo argomento: il numero totale di byte da allocare in memoria. Restituisce poi un puntatore di tipo *void* all'inizio dell'area di memoria allocata.

L'area di memoria allocata non viene posta a 0.

```
int *int_ptr;
...
int_ptr = (int *) malloc(1000 * sizeof(int));
```

5.3 Esaminare il puntatore

Se si richiede più memoria di quella disponibile, le funzioni *calloc()* e *malloc()* restituiscono un **puntatore nullo**. È importante esaminare il puntatore restituito per verificare che l'operazione sia andata a buon fine.

```
int *int_ptr;
...
int_ptr = (int *) calloc(1000, sizeof(int));
if(int_ptr == NULL){
    ...
}
```

Se in questo caso l'allocazione dovesse andare a buon fine, *int_ptr* punterà all'inizio di un array di 1000 interi. È possibile quindi lavorare con questo spazio di memoria come un normale array di interi.

5.4 Funzione free()

La funzione *free()* permette di restituire la memoria allocata dinamicamente attraverso le funzioni *calloc()* e *malloc()*. È importante restituire la memoria allocata dinamicamente quando questa non è più utilizzata, in modo da poter riutilizzare quella porzione di memoria per successive allocazioni.

L'argomento della funzione *free()* è un puntatore all'inizio della memoria allocata (quello restituito dalle funzioni *calloc()* e *malloc()*). È necessario assicurarsi che il puntatore passato sia un valido indirizzo all'inizio della memoria allocata.

```
int_ptr = (int *) calloc(1000, sizeof(int));
...
free(int_ptr);
```

La funzione *free()* non restituisce alcun risultato.

5.5 Matrice dinamica

Per generare una matrice dinamicamente di dimensioni stabilite durante l'esecuzione del programma è necessario avere la seguente struttura:

- Puntatore a una lista
- Lista di puntatori
- Una serie di array-righe raggiungibile dalla precedente lista di puntatori

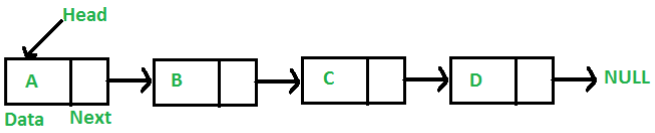
Esempio:

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int i,j;
    int **p;
    int r, c;

    printf("Quante righe e quante colonne?\n");
    scanf("%d %d", &r, &c);

    // array dinamico di puntatori
    p = (int **) malloc(r * sizeof(int *));

    // allocazione delle righe
    for(i=0; i<r; i++)
        p[i] = malloc(c * sizeof(int));

    ...
    ...
    free(p);
    ...

    return 0;
}
```

Liste

Le liste sono delle strutture dati dinamiche che permettono di gestire problemi in cui non si è in grado di prevedere la quantità di dati da salvare in memoria.

Una lista è una struttura dati lineare, in cui gli elementi non sono memorizzati in locazioni di memoria contigue. Gli elementi di una lista sono collegati utilizzando i puntatori come mostrato nell'immagine seguente:



Nello *stack* verrà mantenuto un puntatore che punta alla **testa** della **lista dinamica**. Ogni nodo della lista avrà poi il puntatore all'elemento successivo.

Esempio:

```
#include <stdio.h>
#include <stdlib.h>

struct elem_{
    int num;
    struct elem_ *next;
};
typedef struct elem_ elem;  // rinomino il tipo

elem * inserisciInCoda(elem *, int);
void visualizza(elem *);

int main(){

    return 0;
}
```